Project Handout for DB101 - part 1

Srinath R. Naidu

August 22, 2011

1 Input Description

A graph G(V,E) is given to you. The graph is a complete graph on n vertices and possesses two kinds of edges, of type A and type B. The input for the graph is given to you in the form of a file. The first line is of the form *num_vertices* : *int*. The second to last but one line uses the template : *(vertex1 vertex2) weight*. The last line in the file will be of the form *typeB* : *weight*.

The semantics of the input file are as follows. Each line of the type *(vertex1 vertex2) weight* declares that there is an edge of weight *weight* between vertices *vertex1* and *vertex2*. The type A edges are specified explicitly, whereas type B edges are implicit. Type B edges exist between every pair of vertices between which there is no type A edge. Note that the number of vertices mentioned in the first line of the input description may be larger than the number of vertices mentioned as part of some type A edge. You will need to create identifiers for such vertices in some systematic fashion. Type B edges all have weight given by the figure in the line *type B: weight*.

As mentioned before type A and type B edges on the set of all vertices form a complete graph on the set of all vertices.

2 Tasks

You have three main tasks to be performed on the graph. They are:

- Parse the given input description to create a hash table of type A edges and a hash table and array of vertices, both specified and implicit.

- Scan the hash table to obtain an array of type A edges and then sort the array in ascending order of edge weight. You must use heapsort for this exercise and write all heap-related routines yourself.
- Find the minimum spanning tree of the subgraph of G restricted to type A edges. You must use Kruskal's algorithm.
- Find if the complete graph on type A and type B edges satisfies the *triangle inequality*. The triangle inequality is the requirement that for any three vertices forming a triangle, the sum of the weights on two of the sides of the triangle is greater than the third.

3 Required data structures

We would need to maintain the following data structures.

- A hash table for type A edges. The key for the hash table will need to be computed from the vertex ids of the edge of the graph.
- A hash table that stores the original vertex strings in the input description and the integer-ids created for these vertex strings.
- An array that stores the original vertex strings against the integer-ids created for these vertex strings.
- An array will be needed to store the edges in order of descending weight.
- Kruskal's algorithm will need a union-find set data structure to store vertices belonging to the same sub-tree.

4 Implementation Details

Parsing the input: In order to parse the input description, you need to perform the following sub-tasks:

- read the first line containing the number of vertices.
- read input line into a character string.
- parse the string into tokens representing vertices and edge weight.

- for a given vertex string check the vertex hash table to see if we have already seen this vertex. If we have not seen the vertex, create a new integer id for the vertex and insert the vertex string into an array against the new integer id value. If we have seen the given vertex string, extract its integer id from the hash table. Use chaining for the vertex hash table.
- for each edge seen, create a hash key using the vertex-ids from the previous step. Query the edge hash table for this edge, and if found, update the edge weight as required. If the edge is not found on the edge hash table, create a new edge and store it in the hash table against the key. Use chaining for this hash table.
- if the number of vertices seen so far is less than the original number of vertices specified in the first line, create integer ids for these vertices starting from number of vertices seen so far and create vertex strings such as "user:id" for these vertices. Update both the array and hash table pertaining to vertices.

Performing heap-sort on type A edges: To perform heap-sort on type A edges, follow the steps below:

- Scan the hash table consisting of type A edges and populate an array of edges in arbitrary order. This array will be used to perform in-place sorting using heap-sort.
- Write the procedure MaxHeapify which is the basic procedure used by heaport.
- Write the procedure BuildMaxHeap, which calls MaxHeapify. The pseudo-code for both these procedures is available in CLR.
- Write the procuedure HeapSort which calls BuildMaxHeap and MaxHeapify to sort the given array of edges.

Building the minimum spanning tree: To build the minimum spanning tree, follow the steps below:

- Use sorted array of edges from the previous step.
- Use Kruskal's algorithm to construct the minimum spanning tree. The algorithm proceeds by initially considering all n vertices to be trees in their own right. Then it adds edges between vertices

in ascending order of weight such that either cycles are created, or trees are joined together to create a larger tree. If an edge creates a cycle, it is discarded. Otherwise the edge is retained. The procedure lasts until we add n-1 edges, or we reach a point when we exhaust all edges in the graph.

- In order to decide whether a cycle is created once an edge is added, we maintain a union-find data structure. Briefly, it works as follows: the set of vertices currently in a given tree are all placed in a set data structure. The set is assigned a parent, so that all vertices in the set point to this parent. If an edge is added between two vertices which both point to the same parent, it means that the edge creates a cycle because there already exists a path between those two vertices by virtue of them both being in the same set. Once an edge is added successfully, the two distinct sets that contain the endpoints of the edge are joined together via a union operation.

Checking for the triangle-inequality: In order to check for the triangle inequality, follow the steps below:

- Sort the list of type A edges in descending order. Nothing needs to be done here if the sorted list is already available.
- For each edge from the list of type A edges sorted in descending order, form a triangle with each of the other vertices and check if the triangle inequality is satisfied. In order to get the edge weight for a given edge, query the edge hash table for the given edge. If the edge is available on the hash table, obtain its weight from the hash table. If it is not present on the hash table, it is a type B edge, with weight specified in the input description. The reason for using edges in descending order of weight is so that we can quickly identify cases when the triangle inequality is not satisfied. Notice that it is not necessary to check triangles containing only type B edges, as the triangle inequality is seen to be satisfied for such triangles.

5 Messaging

We expect output in the form of a series of messages indicating what is happening as each of these tasks is executed:

- After parsing we would like a series of messages as below:
  - MSG: number of explicitly specified vertices is ... and total number of vertices is ...
  - : number of type A edges is ...
- After constructing the spanning tree we would like to see a message:
  - Minimum Spanning tree constructed, edges used are ...
  - Cost of minimum spanning tree is ...
- If the spanning tree could not be constructed because the graph on type A edges is disconnected:
  - Global spanning tree could not be constructed, graph has X components.
  - Component 1 has the vertices ..., component 2 has the vertices ..., component X has ...

We will add more messages as the project progresses.